ICS-33: In-Lab Programming Exam #1

Name (printed: Last, First): _____ Lab # (1- 8): _____

Name (signed: First Last): _____ Seat # (1-46): _____

This In-Lab programming exam is worth a total of 100 points. It requires you to define several functions in a module. I will supply a script for calling these functions and printing their results, so that you can check them visually for correctness; we will use a batch self-check file with similar tests for grading purposes only: you will not use **bsc** files. **Although these functions are related, you can write and test these functions in any order. None of the functions rely on each other to work correctly.**

You will have approximately 105 minutes to work on the exam, after logging in and setting up your computer (I estimate that taking about 5 minutes). Make sure that you read the instructions carefully and understand the problems before writing/debugging their code: don't rush through this part of the exam. You may write-on/annotate these pages. Finally, you will write, test, and debug the functions in the **exam.py** module.

We will test your functions only for correctness; each test you pass will improve your grade, and functions that produce no correct results will earn no credit. This means that your functions must define exactly the parameters specified in the descriptions below and must work on all the example arguments; your functions should also work on any other similar arguments. To aid you writing and debugging these functions

1. Write clear, concise, and simple Python code.
2. Choose good names for local variables.

You do not need to include any comments in your code; but, feel free to add them to aid yourself: we will **not** grade you on appropriate variable names or comments. You may also call extra **print** functions in your code or use the Eclipse Debugger to help you understand/debug your functions.

You may use any functions in the standard Python library and the **goody** and **prompt** modules (which will be included in the project file that you will download). Documentation for Python's standard library and the modules from **courselib** will be available during the exam. I have written all the standard **import** statements that I needed in the module in which you will write your functions; feel free to include other imports, or change the form of the included imports to be simpler to use.

If you are having problems with the operating system (logging on, downloading the correct folder/files, accessing the Python documentation, submitting your work) or Eclipse (starting it, setting it up for Python, running Python scripts, running the Eclipse debugger, displaying line numbers) please talk to the staff as soon as possible. We are trying to test only your programming ability, so we will help you with these other activities. But, we cannot help you understand, test, or debug your programming errors.

You should have a good understanding of the solutions to the problems in Programming Assignment #1 (especially the first 3); you should also have a good understanding of the material on Quiz #1. You should know how to read files; create, manipulate, and iterate over lists, tuples, sets, and dictionaries (**dict** and **defaultdict**, 3 ways: keys, values, items); call the functions or methods **all**, **any**; **min**, **max**, **sum**; **join**, **split**; **sort**, **sorted**; **enumerate**, **zip**. Note that you can call **min**, **max**, **sort**, and **sorted** with a **key** function (often a simple **lambda**) that determines how to order the values in the iterable argument on which these functions compute. You are free to use or avoid various Python language features (e.g., lambdas and comprehensions): do what is easiest for you, because we are grading only on whether your functions work correctly.

Summary:

This exam is worth 100 points. It requires you to write five functions according to the specifications below. Each problem is graded on correctness only (how many tests it passes); each problem will be worth 20 points.

The module you will download

1. Defines each method with reasonable annotated parameter names (which you can change) and a body that is **pass** (thus, it returns **None** and will pass no tests).
2. Includes a script that tests these functions individually on different legal inputs, printing the results, and printing whether or not they are correct; if they are not correct, further information is printed.

The next five sections explain the details of these functions. You should probably try writing/testing each function as you read these details. Spend about 15-20 minutes on each function; at that point, if you are not making good progress toward a solution, move on to work on later functions (which you might find easier) and return if you have time. Each problem is graded based on the percentage of tests it passes.

A Data Structure for Phone Calls

Suppose that the US government, for security reasons, wanted to store a database containing basic information about every telephone call made. We will represent this information in a **dictionary** whose keys specify the **caller** (the person who placed the call). Associated with each **caller** is the another **dictionary** whose keys specify the **receiver** (the person receiving the calls); associated with each **receiver** is a **list** of non-negative **int** values: each specifies the amount of time spent (in minutes) for each call that the **caller** placed to the **receiver**.

For example a simple/small database might be.

```
db = {'Alan': {'Barb': [10, 20], 'Carl': [10, 10]},
      'Barb': {'Alan': [10, 10], 'Deja': [5, 5, 5]},
      'Carl': {'Alan': [10, 5],  'Deja': [15],       'Elan': [5, 5]},
      'Deja': {'Elan': [5]},
      'Elan': {'Carl': [10]}}
```

I have printed this dictionary in an easy to read form. Python prints dictionaries on one line, and can print their key/values in any order, because dictionaries are not ordered.

This data structure means that

1. **Alan** called **Barb** twice (**10** and **20** minute calls) and called **Carl** twice (**10** and **10** minute calls).
2. **Barb** called **Alan** twice (**10** and **10** minute calls) and called **Deja** three times (**5**, **5**, and **5** minute calls).
3. **Carl** called **Alan** twice (**10** and **5** minute calls), called **Deja** once (**15** minute call), and called **Elan** twice (**5** and **5** minute calls).
4. **Deja** called **Elan** once (**5** minute call).
5. **Elan** called **Carl** once (**10** minute call).

1: Details of all_parties:

The **all_parties** function takes a **dict** argument in the form illustrated on the middle of page 2,

> each key in the **dict** is a **str** (the **caller**) whose associated value is another **dict**: this inner **dict** associates a **str** key (the **receiver**) with a list of **int** values (the **talk times**) specifying the amount of time spent (in minutes) for each call that the **caller** placed to the **receiver**.

and returns a **set** of all names (**callers** or **receivers**) who participated in any phone calls. **IMPORTANT**: In some databases, a person might not appear as a **caller**, but only appear as a **receiver**: this person should appear

in the **set** returned by this function. The parameter dictionary should not be changed.

Calling this function with **db** bound to the dictionary shown on the middle of page 2: **all_parties(db)**

returns the following **set**:

```
{'Deja', 'Barb', 'Elan', 'Carl', 'Alan'}
```

Python prints set elements in any order, because sets are not ordered.

2: Details of read_db:

The **read_db** function takes an open file as an argument; it returns a **dict** in the form illustrated on the middle of page 2.

The input file will consist of some number of lines; each line specifies a caller, followed by a receiver, followed by one or more **ints**, each representing the length of a call placed from the caller to receiver. All these different items are separated by colons (**:**) and contain no internal spaces.

For example, the **db1.txt** file contains the lines

```
Alan:Barb:10:20
Alan:Carl:10:10
Barb:Alan:10:10
Barb:Deja:5:5:5
Carl:Alan:10:5
Carl:Deja:15
Carl:Elan:5:5
Deja:Elan:5
Elan:Carl:10
```

I have listed all calls made by the same caller together, but you function should be able to read a file with these same lines in any order and produce the correct result.

For this file, **read_db** returns a dictionary whose contents are:

```
{'Alan': {'Barb': [10, 20], 'Carl': [10, 10]},
 'Barb': {'Alan': [10, 10], 'Deja': [5, 5, 5]},
 'Carl': {'Alan': [10, 5],  'Deja': [15],       'Elan': [5, 5]},
 'Deja': {'Elan': [5]},
 'Elan': {'Carl': [10]}}
```

Important: Remember to convert each **time** from a **str** into an **int**

I have printed this dictionary in an easy to read form. Python prints dictionaries on one line, and can print their key/values in any order, because dictionaries are not ordered.

3: Details of talker_frequency:

The **talker_frequency** function takes a **dict** argument in the form illustrated on the middle of page 2,

> each key in the **dict** is a **str** (the **caller**) whose associated value is another **dict**: this inner **dict** associates a **str** key (the **receiver**) with a list of **int** values (the **talk times**) specifying the amount of time spent (in minutes) for each call that the **caller** placed to the **receiver**.

and returns a **list** of **2-tuples**: the first index is a person's name and the second index is the number of calls the person participated in (as either **caller** or **receiver**). The parameter dictionary should not be changed. The list

elements appear in **decreasing order** of the number of calls: if two names have the same number of calls, they appear in **increasing alphabetic** order.

Calling this function with the **db** dictionary shown on the middle of page 2: **talker_frequency(db)**

returns the following :

```
[('Alan', 8), ('Carl', 8), ('Barb', 7), ('Deja', 5), ('Elan', 4)]
```

Notice that **Alan** and **Carl** participate in the same number of calls (**8**), so they appear in increasing alphabetical order.

4: Details of mutual_callers:

The **mutual_callers** function takes a **dict** argument in the form illustrated on the middle of page 2,

> each key in the **dict** is a **str** (the **caller**) whose associated value is another **dict**: this inner **dict** associates a **str** key (the **receiver**) with a list of **int** values (the **talk times**) specifying the amount of time spent (in minutes) for each call that the **caller** placed to the **receiver**.

and returns a **set** of **2-tuples**: both indexes in the **2-tuple** are names; these name pairs must appear in increasing alphabetical order. Two names are in a **tuple** in the **set** if the the first has placed one or more calls to the second, **and** the second name has placed one or more calls to the first: they have called each other. The parameter dictionary should not be changed.

Calling this function with the **db** dictionary shown on the middle of page 2: **mutual_callers(db)**

returns a set whose elements are:

```
{('Alan', 'Carl'), ('Carl', 'Elan'), ('Alan', 'Barb')}
```

Python prints set elements in any order, because sets are not ordered.

Reminder: The names in each **2-tuple** must appear in increasing alphabetical order.

5: Details of initiators:

The **initiators** function takes a **dict** argument in the form illustrated on the middle of page 2,

> each key in the **dict** is a **str** (the **caller**) whose associated value is another **dict**: this inner **dict** associates a **str** key (the **receiver**) with a list of **int** values (the **talk times**) specifying the amount of time spent (in minutes) for each call that the **caller** placed to the **receiver**.

and returns a **list** of names (**str**). The order of the names is (a) **decreasing** by number of times they are the **caller**; if two names are a **caller** the same number of times, the order is (b) **increasing** by number of times they are the **receiver**; and if two names are both a **caller** and **receiver** the same number of times, they appear in (c) **increasing alphabetical** order by name. The parameter dictionary should not be changed.

Calling this function with the **db** dictionary shown on the middle of page 2: **initiators(db)**

returns the following **list**:

```
['Barb', 'Carl', 'Alan', 'Elan', 'Deja']
```

Why this order? We can compute that

1. **Barb** is the **caller 5** times and the **receiver 2** times
2. **Carl** is the **caller 5** times and the **receiver 3** times.
3. **Alan** is the **caller 4** times and the **receiver 4** times.
4. **Elan** is the **caller 1** time and the **receiver 3** times.
5. **Deja** is the **caller 1** time and the **receiver 4** times.

So, **Barb** precedes **Carl** because although each are the **caller 5** times, **Barb** is the **receiver** fewer times: so, by (b) **Barb** comes earlier.

6: Details of matrix (Extra Credit: 1 point; finish all previous problems first):

The **matrix** function takes a **dict** argument in the form illustrated on the middle of page 2,

> each key in the **dict** is a **str** (the **caller**) whose associated value is another **dict**: this inner **dict** associates a **str** key (the **receiver**) with a list of **int** values (the **talk times**) specifying the amount of time spent (in minutes) for each call that the **caller** placed to the **receiver**.

and returns a **list** of **list** of (**int**). The parameter dictionary should not be changed.

Calling this function with the **db** dictionary shown on the middle of page 2: **matrix(db)**

returns the **list** of **list**:

```
[[0, 50, 35, 0, 0],
 [50, 0, 0, 15, 0],
 [35, 0, 0, 15, 20],
 [0, 15, 15, 0, 5],
 [0, 0, 20, 5, 0]]
```

Here is how to interpret this **list** of **list** (see below for the meaning of the cell marked with a *****; of course the ***** does not appear in the list of lists). Basically each cell shows for how many minutes the the row and column people spoke to each other.

```
      Alan Barb Carl Deja Elan
----+-----+----+----+----+----+
Alan|   0 | 50 | 35 |  0 |  0 |
----+-----+----+----+----+----+
Barb|  50 |  0 |  0 | 15 |  0 |
----+-----+----+----+----+----+
Carl|  35 |  0 |  0 | 15*| 20 |
----+-----+----+----+----+----+
Deja|   0 | 15 | 15 |  0 |  5 |
----+-----+----+----+----+----+
Elan|   0 |  0 | 20 |  5 |  0 |
----+-----+----+----+----+----+
```

The horizontal and vertical names appear in the same order: the names of all the **callers** and **receivers** in alphabetical order. The cells at the intersection between the name **N1** in a row and the name **N2** in a column shows for how many minutes **N1** spoke with **N2**. So, the cell marked with a ***** indicates **Carl** (the row name) spoke with **Deja** (the column name) for 15 minutes. The cell for row **Deja** and column **Carl** will be the same, because **Deja** spoke to **Carl** for the same number of minutes that **Carl** spoke to **Deja**.

Since no one can call themself, we are guaranteed to store **0** values on the main diagonal: for the same name in a row and in a column; there are other zeroes because other people did not speak (e.g., **Alan** and **Elan**).